Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors

Ling Ren, Xiangyao Yu, Christopher W. Fletcher *, Marten van Dijk and Srinivas Devadas
MIT CSAIL, Cambridge, MA, USA
{renling, yxy, cwfletch, marten, devadas}@mit.edu

ABSTRACT

Keeping user data private is a huge problem both in cloud computing and computation outsourcing. One paradigm to achieve data privacy is to use tamper-resistant processors, inside which users' private data is decrypted and computed upon. These processors need to interact with untrusted external memory. Even if we encrypt all data that leaves the trusted processor, however, the address sequence that goes off-chip may still leak information. To prevent this address leakage, the security community has proposed ORAM (Oblivious RAM). ORAM has mainly been explored in server/file settings which assume a vastly different computation model than secure processors. Not surprisingly, naïvely applying ORAM to a secure processor setting incurs large performance overheads.

In this paper, a recent proposal called Path ORAM is studied. We demonstrate techniques to make Path ORAM practical in a secure processor setting. We introduce background eviction schemes to prevent Path ORAM failure and allow for a performance-driven design space exploration. We propose a concept called super blocks to further improve Path ORAM's performance, and also show an efficient integrity verification scheme for Path ORAM. With our optimizations, Path ORAM overhead drops by 41.8%, and SPEC benchmark execution time improves by 52.4% in relation to a baseline configuration. Our work can be used to improve the security level of previous secure processors.

*Christopher Fletcher was supported by a National Science Foundation Graduate Research Fellowship, Grant No. 1122374, and a DoD National Defense Science and Engineering Graduate Fellowship. This research was partially supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract N66001-10-2-4089. The opinions in this paper don't necessarily represent DARPA or official US policy.

ISCA '13 Tel-Aviv, Israel

1. INTRODUCTION

1.1 Motivation

Security of private data when outsourcing computation to an untrusted server is a huge security problem. When an untrusted server receives private data from a user, the typical setup places the private/encrypted data along with the program in a tamper-proof environment (e.g., secure processor or coprocessor attached to the server), at which point the data is decrypted and the program is run [13, 23]. Secure coprocessors such as Aegis [23] and XOM [13] or Trusted Platform Module (TPM) based systems generally assume the program being run is trusted–that is, not intentionally malicious and believed to be free of bugs that could leak information about the private data, through pin traffic for example.

Having a trusted program is a lofty and sometimes impractical assumption; the program's source code may be complex or even hidden and therefore not certified by the user. For example, the user may request that a program (e.g., medical diagnosis software) be run on the data, but not know the details of the software. The server itself may be honest-but-curious or even malicious. Security is broken when the server can learn something about the private data by applying the client program or some other curious program to the encrypted data and monitoring the secure processor's side channels such as external pin traffic.

We note that some efforts to limit leakage through memory access patterns (e.g., HIDE [30]) have applied random shuffling to small chunks of memory. While HIDE and related techniques are quite efficient, obfuscation over small chunks does not achieve security when the untrusted server specifies the client program (see Section 6.2).

Completely stopping information leakage through memory access patterns requires the use of Oblivious RAMs (ORAMs) [7, 15, 8]. ORAMs make the sequence of memory locations accessed indistinguishable from a random sequence of accesses, from a cryptographic standpoint. There has been significant follow-up work that has resulted in more efficient ORAM schemes [19, 21]. But till recently, ORAM has been assumed to be too expensive to integrate into a processor from a performance overhead standpoint.

Our focus in this paper is on Path ORAM, a recent ORAM construction introduced in [20]. A recently-proposed secure processor architecture called *Ascend* performs encrypted computation assuming untrusted programs and furthermore uses Path ORAM to obfuscate memory access patterns [3, 4]. We use the Path ORAM configuration in [3] as a baseline

in this paper. Our focus here is on optimizing the Path ORAM *primitive* in a secure processor setting, so it can be more efficiently integrated into all types of secure processors, including Ascend.

1.2 Our Contribution

We believe ORAM is a useful cryptographic primitive in many secure architecture settings, but it has not received much attention from the architecture community thus far. In this paper, we make the following contributions:

1. We present optimizations to Path ORAM to make it more suitable for implementation in a secure processor setting;
2. We give a provably-secure *background eviction scheme* that prevents so-called *Path ORAM failure* (defined in Section 2) and enables more efficient ORAM configurations;
3. We shrink the dimensions of failure probability and performance overhead to a single dimension, allowing for easy design space exploration;
4. We propose the notion of super blocks, further improving Path ORAM performance;
5. We show how to efficiently implement Path ORAM on commodity DRAM;
6. We show that combining all of our optimizations results in a 41.8% reduction in Path ORAM overhead and a 52.4% improvement on SPEC benchmarks execution time in relation to a baseline Path ORAM configuration; and
7. We propose an efficient integrity verification layer for Path ORAM.

Integrity verification [6] and encryption [13] [23] of memory contents were initially considered difficult to do without serious performance degradation prior to architectural research (e.g., [22], [28], [26], [10]) that addressed processor performance bottlenecks. We take a similar first step for Oblivious RAM in this paper.

1.3 Paper Organization

We give background on ORAM and describe Path ORAM in Section 2. Our improvements to Path ORAM are described in Section 3, and evaluation results are provided in Section 4. Section 5 introduces a efficient integrity verification scheme for Path ORAM. Related work is described in Section 6, and we conclude the paper in Section 7.

2. OBLIVIOUS RAM

Suppose we are given program P with input M and any other program P' with input M' and compare the first T memory requests made by each (denoted $\mathsf{transcript}_T(P(M))$ and $\mathsf{transcript}_T(P'(M'))$). A transcript is a list of requests: each request is composed of an address, operation (read or write) and data (if the operation is a write). Oblivious RAM (ORAM) guarantees that $\mathsf{transcript}_T(P(M))$ and $\mathsf{transcript}_T(P'(M'))$ are computationally indistinguishable. Crucially, this is saying that the access pattern is independent of the program and data being run.

A simple ORAM scheme that satisfies the above property is to read/write the entire contents of the program memory to perform every load/store. To hide whether a particular block was needed in the memory scan (and if it was, whether the operation was a read or a write), every block must be encrypted using randomized encryption (e.g., AES in CTR mode), which means that with overwhelming probability the bitstring making up each block in memory will change. With this scheme, the access pattern is independent of the program or its data but clearly it will have unacceptable overheads (on order the size of the memory). Modern ORAM schemes achieve the same level of security through being probabilistic. In this work, we focus on a recent proposal called Path ORAM [20] because of its practical performance and simplicity.

ORAM assumes that the adversary sees $\mathsf{transcript}_T(P(M))$ as opposed to other program state for $P(M)$. A trusted ORAM client algorithm, which we refer to as the *ORAM interface*, translates program memory requests into random-looking requests that will be sent to an untrusted external memory where data is actually stored. (In our secure processor setting, the ORAM interface is analogous to a memory controller.) Note that the ORAM interface's job is only to protect against leakage through $\mathsf{transcript}_T(P(M))$, given a T fixed across all transcripts. If the adversary compares two transcripts of different length, clearly the adversary can tell them apart. Furthermore, *when* each access in the transcript is made can leak information. Ascend [3, 4] deals with these leakage channels by forcing periodic requests of ORAM and predetermined program running time. However, this paper will focus only on making the ORAM primitive as efficient as possible since it is a least-common-denominator in any scheme.

2.1 Basic Path ORAM

In Path ORAM, the external memory is structured as a balanced binary tree, where each node is a *bucket* that can hold up to Z blocks. The root is referred to as level 0, and the leaves as level L. This gives a tree with $L+1$ levels, holding up to $N = Z(2^{L+1} - 1)$ data blocks (which are analogous to processor cache lines in our setting). The remaining space is filled with *dummy blocks* that can be replaced with real blocks as needed. As with data blocks in the naïve memory scan scheme, each block in the ORAM tree is encrypted with randomized encryption.

The ORAM interface for Path ORAM is composed of two main structures, a *stash*[1] and a *position map*, and associated control logic. The position map is an N-entry lookup table that associates the program address of each data block with a leaf in the ORAM tree. The stash is a memory that stores up to a small number, C, of data blocks from the ORAM tree at a time. Now we describe how Path ORAM works. Readers can refer to [20] for a more detailed description.

At any time, each data block stored in the ORAM is mapped (at random) to one of the 2^L leaves in the ORAM tree via the position map (i.e., $\forall$ leaves l and blocks b, $\mathsf{Prob}(b \text{ is mapped to } l) = 1/2^L$). Path ORAM's invariant (Figure 1) is: *If l is the leaf currently assigned to some block b, then b is stored (a) in some bucket on the path from the root of the ORAM tree to leaf l, or (b) in the stash within the ORAM interface.* The path from the root to leaf l is also referred to as path l.

Initially, the ORAM is empty and the position map associates each program address with a random leaf. Suppose a program wants to access some block b with program address u. It makes the request through the ORAM interface via $\mathsf{accessORAM}(u, op, b')$:

[1]This is the local cache in [20]. We changed the term to distinguish it from a processor's on-chip cache.

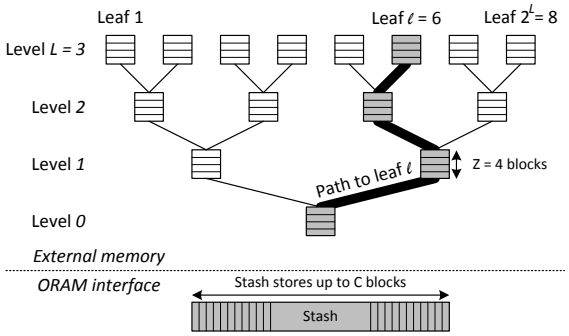

Figure 1: A Path ORAM of $L = 3$ levels. At any time, blocks mapped to leaf $l = 6$ can be located in any of the shaded structures (i.e., on path 6 or in the stash).

1. Look up the position map with u, yielding leaf label l.
2. Read and decrypt all the blocks along path l. Add all the real blocks to the stash. Path ORAM's invariant guarantees that if block b exists, it must be in the stash at this point.
3. If $op = read$, return b if it exists; otherwise return nil. If $op = write$, replace b with b' if it exists; otherwise add a new block b' to the stash.
4. Replace l with a new randomly-selected label l'.
5. Evict and encrypt as many blocks from the updated stash into path l in the ORAM tree. If there is space in any of the buckets along the path that cannot be filled with data blocks, fill that space with encryptions of dummy blocks.

We will refer to steps 2-5 as $\mathsf{accessPath}(u, l, l', op, b')$ later in the paper.

On a path read, all the blocks (including dummy blocks) are read and decrypted, but only real blocks are stored into the stash. For example in Figure 2, the dummy block in leaf 3 is not put into the stash. Address $u = 0$ is reserved for dummy blocks.

Step 4 is the key to Path ORAM's security: whenever a block is accessed, that block is randomly *remapped* to a new leaf in the ORAM tree (see Figure 2 for an example). accessORAM() leaks no information on the address accessed, because a randomly selected path is read and written on every access regardless of the program memory address sequence. Furthermore, since data/dummy blocks are put through randomized encryption, the attacker will not be able to tell which block (if any) along the path is actually needed.

Step 5 is the ORAM 'shuffle' operation from the literature [8]. The idea is that as blocks are read into the stash, in order to keep the stash size small, step 5 tries to write as many blocks to the tree as possible, and tries to put each block as close to the leaves as possible. In the top right box (# 3) in Figure 2: $(b, 1)$ can only go to the root bucket since it only shares the root bucket in common with the path 3; $(c, 2)$ can no longer be written back to the tree at all since it only shares the root bucket with path 3, and the root bucket is now full; $(d, 4)$ can be mapped back to the bucket between the leaf and the root; no block goes to leaf 3, so that bucket needs to be filled up with the encryption of a dummy block. After all of the above computation is done, the ORAM interface writes back the path in a data-independent order (e.g., from the root to the leaf).

2.2 Randomized Encryption for Path ORAM

The Path ORAM tree and stash have to store a (leaf, program address, data) triplet for each data block. Let B be the data block size in bits. Each leaf is labeled by L bits and the program address is stored in $U = \lceil \log_2 N \rceil$ bits. Then each bucket contains $Z(L + U + B)$ bits of plaintext. As mentioned, the protocol requires randomized encryption over each block (including dummy blocks) in external memory, adding extra storage to each bucket. We first introduce a strawman randomized encryption scheme, and then propose a counter-based randomized encryption scheme to reduce the bucket size.

2.2.1 Strawman scheme

A strawman scheme to fully encrypt a bucket (used in [3]) is based on AES-128: On a per-bucket basis, apply the following operation to each block in the bucket:

1. Generate a random 128-bit key K' and encrypt K' using the processor's secret key K (i.e., $\mathsf{AES}_K(K')$).
2. Break up the B plaintext bits into 128-bit chunks (for AES) and apply a one-time-pad (OTP) to each chunk that is generated through K' (i.e., to encrypt $chunk_i$, we form the ciphertext $\mathsf{AES}_{K'}(i) \oplus chunk_i$).

The encrypted block is the concatenation of $\mathsf{AES}_K(K')$ and the OTP chunks, and the encrypted bucket is the concatenation of all of the Z encrypted blocks. Thus, this scheme gives a bucket size of $M = Z(128 + L + U + B)$ bits where $Z(L+U+B)$ is the number of plaintext bits per bucket from the previous section. Note that since we are using OTPs, each triplet of $(L + U + B)$ bits does not have to be padded to a multiple of 128 bits.

2.2.2 Counter-based scheme

The downside to the strawman scheme is the extra 128 bits of overhead per block that is used to store $\mathsf{AES}_K(K')$. We can reduce this overhead by a factor of $2 \cdot Z$ by introducing a 64-bit counter per bucket (referred to as $BucketCounter$). To encrypt a bucket:

1. $BucketCounter \leftarrow BucketCounter + 1$.
2. Break up the plaintext bits that make up the bucket into 128-bit chunks. To encrypt $chunk_i$, apply the following OTP: $\mathsf{AES}_K(BucketID||BucketCounter||i) \oplus chunk_i$, where $BucketID$ is a unique identifier for each bucket in the ORAM tree.

The encrypted bucket is the concatenation of each chunk along with the $BucketCounter$ value in the clear. $BucketCounter$ is set to 64 bits so that the counter will not roll over. $BucketCounter$ does not need to be initialized; it can start with any value.

This scheme works due to the insight that buckets are always read/written atomically. Seeding the OTP with $BucketID$ is important: it ensures that two *distinct* buckets in the ORAM tree will not have the same OTP. A new random key K is picked each time a program starts, so that the OTPs used across different runs will be different to defend replay attacks. With this scheme, $M = Z(L+U+B)+64$ bits which we assume for the rest of the paper.

2.3 Hierarchical Path ORAM

The $N \cdot L$-bit position map is usually too large, especially for a secure processor's on-chip storage. For example, a 4 GB Path ORAM with a block size of 128 bytes and $Z = 4$ has a position map of 93 MB. The hierarchical Path ORAM

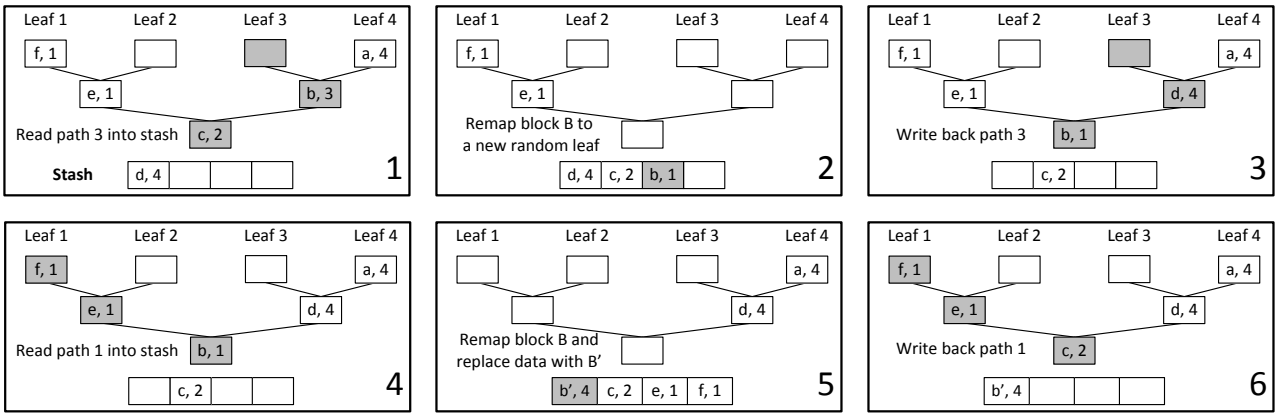

Figure 2: An example Path ORAM operation ($Z = 1$ and $C = 4$) for reading block b and then updating b to be b'. Each block has the format: 'block identifier','leaf label' (e.g., $(b, 3)$). If the program address for b is u, steps 1-3 correspond to accessORAM$(u, read, -)$ **and steps 4-6 are** accessORAM$(u, write, b')$ **from Section 2.1. An adversary only sees the ORAM interface read/write two random paths (in this case path 3 and path 1).**

addresses this problem by storing the position map in an additional ORAM (this idea was first mentioned in [21]).

We will refer to the first ORAM in a hierarchy as the data (Path) ORAM or $ORAM_1$. $ORAM_1$'s position map will now be stored in a second ORAM $ORAM_2$. If $ORAM_2$'s position map is still too large, we can repeat the process with an $ORAM_3$ or with however many ORAMs are needed. $ORAM_i, (i \geq 1)$ are referred to as *position map ORAMs*. To perform an access to the data ORAM in a hierarchy of H ORAMs, we first look up the on-chip position map for $ORAM_H$, then perform an access to $ORAM_H, ORAM_{H-1}, \ldots, ORAM_1$. Each ORAM lookup yields the path to access in the next ORAM.

To be concrete, we give an example with a 2-level hierarchical Path ORAM. Let N_h, L_h, B_h, M_h, C_h, and Z_h be the parameters for $ORAM_h$ $(h = 1, 2$, variable names are analogous to Section 2.1). Since the position map of $ORAM_1$ has N_1 entries and each block in $ORAM_2$ is able to store $k_2 = \lfloor B_2/L_1 \rfloor$ labels, $ORAM_2$'s capacity must be at least $N_2 = \lceil N_1/k_2 \rceil \approx N_1 \cdot L_1/B_2$. The number of levels in $ORAM_2$ is $L_2 = \lceil \log_2 N_2 \rceil - 1$.

The invariant is, if some data block b_1 in $ORAM_1$ has program address u_1, then

1. there exists a block b_2 in $ORAM_2$ with program address $u_2 = \lfloor u_1/k_2 \rfloor + 1$[2]; the $i = u_1 - (u_2 - 1)k_2$-th leaf label stored in b_2 equals l_1.
2. For $h = 1, 2$, b_h is mapped to a uniformly random leaf $l_h \in \{1, \ldots, 2^{L_h}\}$ in $ORAM_h$'s tree; b_h is either in some bucket along path l_h in $ORAM_h$'s tree, or in $ORAM_h$'s stash (the Path ORAM invariant holds for each ORAM in the hierarchy).

Given the above invariants, accessHORAM(u_1, op, b_1') below describes a complete 2-level hierarchical ORAM access:

1. Generate random leaf labels l_1' and l_2'. Determine i and u_2 as described in the invariant.
2. Lookup $ORAM_2$'s position map with u_2, yielding l_2.
3. Perform accessPath$(u_2, l_2, l_2', write, b_2')$ on $ORAM_2$, yielding block b_2 (as described in the invariant). Record l_1, the ith leaf label in b_2. Replace l_1 with l_1' to get b_2'.
4. Perform accessPath$(u_1, l_1, l_1', op, b_1')$ on $ORAM_1$. This completes the operation.

accessPath() is defined in Section 2.1.

A hierarchical Path ORAM requires an additional state machine to decide which ORAM is being accessed and requires additional storage for each ORAM's stash.

2.4 Path ORAM Storage & Access Overhead

To store up to $N \cdot B$ data bits, the Path ORAM tree uses $(2^{L+1} - 1) \cdot M$ bits, where $M = Z(L + U + B) + 64$ as defined in Section 2.2.2. In practice, the Path ORAM tree would be stored in DRAM. In that case M should be rounded up to a multiple of DRAM access granularity (e.g. 64 bytes). For a hierarchical ORAM with H ORAMs, the on-chip storage includes the stash for each ORAM, $\sum_{i=1}^{H} C_i(L_i + U_i + B_i)$ bits in total, and the $N_H \cdot L_H$-bit position map for $ORAM_H$.

We define Path ORAM *Access_Overhead* as the ratio between the amount of data moved and the amount of useful data per ORAM access. In order to access B bits (one data block) in Path ORAM, $(L + 1)M$ bits (an entire path) have to be read and written, giving $Access\_Overhead = \frac{2(L+1)M}{B}$. The access overhead of hierarchical Path ORAM is similarly defined as $\frac{\sum_{i=1}^{H} 2(L_i+1)M_i}{B_1}$. The denominator is B_1 because only the block in data ORAM is needed by the processor.

2.5 Limitations of Path ORAM for Secure Processors

As mentioned, Path ORAM was not originally designed for secure processors. Below we list the limiting factors for Path ORAM in a secure processor setting, and briefly talk about how we will address them.

2.5.1 Stash overflow

Path ORAM fails when its stash overflows. Despite the write-back operation, blocks can still accumulate in Path ORAM's stash. When a block is remapped on an access, the probability that it can be written back to the same path is low. This may cause the total number of blocks in the stash to increase by one after an access.

[2]The +1 offset is because address $u_2 = 0$ is reserved for dummy blocks in $ORAM_2$.

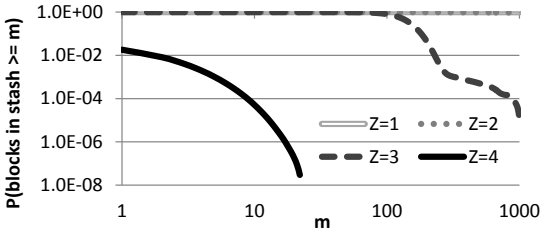

Figure 3: The probability that the number of blocks in the stash exceeds a certain threshold for different Z, in a 4 GB Path ORAM with 2 GB working set and an infinitely large stash.

Figure 3 gives the cumulative distribution of stash occupancy for a 4 GB Path ORAM with 2 GB working set and different Z, assuming an infinitely large stash. In this experiment, we take a data point after every access and show the histogram. In particular, Figure 3 shows the probability that the number of blocks in the stash exceeds a certain threshold C, which is equivalent to the failure probability with a stash of size C. Even with a stash size of 1000 blocks, Path ORAM with $Z \leq 2$ always fails and Path ORAM with $Z = 3$ fails with $\sim 10^{-5}$ probability. This problem can be alleviated by making $Z \geq 4$. However, a larger Z increases *Access_Overhead* (Section 2.4). We will introduce *background eviction*—a scheme to empty the stash when it fills—to eliminate Path ORAM failure probability in Section 3.1.

2.5.2 Access latency

To access a data block/cache line, a normal processor will initiate a fast page or burst command to a DRAM to access specifically the cache line of interest (B bits). By contrast, Path ORAM moves hundreds of times more data (given by *Access_Overhead*) than a normal processor does per access, significantly increasing the memory access latency.[3]

Decreasing Z can proportionally decrease Path ORAM's access overhead. However, such configurations are precluded because they significantly increase the failure probability (Figure 3). In Section 4.1 we will use our background eviction technique to make these higher performance configurations possible in a secure processor setting.

2.5.3 Low DRAM utilization

In the original Path ORAM paper [20], the authors suggest setting the maximum number of data blocks in the ORAM to the number of buckets, which means only $1/Z$ of blocks contain valid data (the rest is made up of dummy blocks). As with trying to reduce latency, increasing the number of valid blocks in the ORAM, while keeping the ORAM capacity and Z fixed, leads to a larger failure probability. In Section 4.1.3 we show DRAM utilization can be improved.

3. PATH ORAM OPTIMIZATION

We now describe techniques to improve Path ORAM in a secure processor context.

[3]In fact, Path ORAM latency accounts for most of the performance overhead in the Ascend secure processor [3, 4].

3.1 Background Eviction

3.1.1 Proposed background eviction scheme

To be usable, a background eviction scheme must (a) not change the ORAM's security guarantees, (b) make the probability of stash overflow negligible and (c) introduce as little additional overhead to the ORAM's normal operation as possible. For instance, a strawman scheme could be to read/write every bucket in the ORAM tree when stash occupancy reaches a threshold—clearly not acceptable from a performance standpoint.

Unfortunately, the strawman scheme is also not secure. We make a key observation that if background evictions occur when stash occupancy reaches a threshold, the fact that background evictions occurred can leak privacy because some access patterns fill up the stash faster than others. For example, if a program keeps accessing the same block over and over again, the requested block is likely to be already in the stash—not increasing the number of blocks in the stash. In contrast, a program that scans the memory (i.e., accesses all the blocks one by one) fills up the stash much faster. If an attacker realizes that background evictions happen frequently, the attacker can infer that the access pattern of the program is similar to a memory scan and can possibly learn something about private data based on the access pattern.

One way to prevent attacks based on *when* background evictions take place is to make background evictions *indistinguishable* from regular ORAM accesses. *Our proposed background eviction scheme prevents Path ORAM stash overflow using dummy load/stores.* To prevent stash overflow, we stop serving real memory requests and issue dummy requests whenever the number of blocks in the stash exceeds $C - Z(L+1)$. (Since there can be up to $Z(L+1)$ real blocks on a path, the next access has a chance to overflow the stash at this point.) A dummy access reads and decrypts a random path and writes back (after re-encryption) as many blocks from the path and stash as possible. A dummy access will at least not add blocks to the stash because all the blocks on that path can at least go back to their original places (note that no block is remapped on a dummy access). Furthermore, there is a possibility that some blocks in the stash will find places on this path. Thus, the stash cannot overflow and Path ORAM cannot fail, by the definition of ORAM failure we have presented so far, with our background eviction scheme. We keep issuing dummy accesses until the number of blocks in the stash drops below the $C - Z(L+1)$ threshold, at which point the ORAM can resume serving real requests again.

Our background eviction scheme can be easily extended to a hierarchical Path ORAM. If the stash of any of the ORAMs in the hierarchy exceeds the threshold, we issue a dummy request to each of the path ORAMs in the same order as a normal access, i.e., the smallest Path ORAM first and the data ORAM last.

Livelock. Our proposed background eviction scheme does have an extremely low probability of *livelock*. Livelock occurs when no finite number of background evictions is able to reduce the stash occupancy to below $C - Z(L+1)$ blocks. For example, all blocks along a path may be mapped to the same leaf l and every block in the (full) stash might also map to leaf l. In that case no blocks in the stash can be evicted, and dummy accesses are continually performed (this is similar to a program hanging). However, the possibility of such a

scenario is similar to that of randomly throwing 32 million balls (blocks) to 16 million bins (leafs) with more than 200 balls (stash size) landing into the same bin—astronomically small (on the 10^{-100} scale). Therefore, we do not try to detect or deal with this type of livelock. We note that livelock does not compromise security.

3.1.2 Security of the proposed background eviction

Our background eviction scheme does not leak any information. Recall that the original Path ORAM (with an infinite stash and no background eviction) is secure because, independent of the memory requests, an observer sees a sequence of random paths being accessed, denoted as

$$P = \{p_1, p_2, \ldots, p_k, \ldots\},$$

where p_k is the path that is accessed on kth memory access. Each $p_k, (k = 1, 2, \ldots)$ follows a uniformly random distribution and is independent of any other p_j in the sequence. Background eviction interleaves another sequence of random paths q_m caused by dummy accesses, producing a new sequence

$$Q = \{p_1, p_2, \ldots, p_{k_1}, q_1, \ldots, p_{k_2}, q_2, \ldots\}.$$

Since q_m follows the same uniformly random distribution with p_k, and q_m is independent of any p_k and any $q_n (n \neq m)$, Q also consists of randomly selected paths, and thus is indistinguishable from P. This shows the security of the proposed background eviction.

3.1.3 Examples of insecure eviction schemes

We point out that attempts to eliminate livelock (Section 3.1.1) are likely to break security. We examine the following potentially insecure eviction scheme: When the number of blocks in the stash reaches the threshold, we randomly access a block that is in the stash (referred to as the *block remapping scheme*). This scheme will not livelock because the blocks in the stash will gradually get remapped and 'escape' the congested path. Unfortunately, this is also why security breaks.

We first define $\mathsf{CPL}(p, p')$, the Common Path Length of path p and p', which is the number of buckets shared by the two paths. Given arbitrary p and p', $\mathsf{CPL}(p, p')$ may be between 1 and $L + 1$ (two paths at least share the root bucket, and there are $L + 1$ levels in total). Using Figure 1 as an example, $\mathsf{CPL}(1, 2) = 3$ and $\mathsf{CPL}(3, 8) = 1$. Given an ORAM tree of $L + 1$ levels, if p and p' are drawn from a uniform distribution, then

$$P\left(\mathsf{CPL}(p, p') = l\right) = \left\{ \begin{array}{l} \frac{1}{2^l},\ 1 \leq l \leq L \\ \frac{1}{2^L},\ l = L + 1 \end{array} \right\},$$

$$E\left[\mathsf{CPL}(p, p')\right] = 2 - \frac{1}{2^L}.$$

For the proposed secure background eviction scheme, the average CPL should be very close to the expectation. However, for the sequence Q with the block remapping eviction scheme, each q_m (an access for eviction) is the leaf label of block u_m that is in the stash at that point. Note that a block mapped to path p is less likely to be evicted to the ORAM tree if the accessed path p' shares a shorter common path with p. Therefore, the fact that block u_m is in the stash suggests that the access prior to it, which is p_{k_m}, is not in favor of the eviction of u_m. So if this eviction scheme is used, the average CPL between consecutive paths in Q will be significantly smaller than the expected value $2 - \frac{1}{2^L}$. We mount this attack 100 times on a Path ORAM with $L = 5$, $Z = 1$ and $C - Z(L + 1) = 2$. Figure 4 shows that our attack can detect the insecure evictions. The average CPL of our proposed background eviction scheme is 1.979, very close to the expected value $2 - \frac{1}{2^5} \approx 1.969$, while the average CPL of the insecure eviction scheme is 1.79.

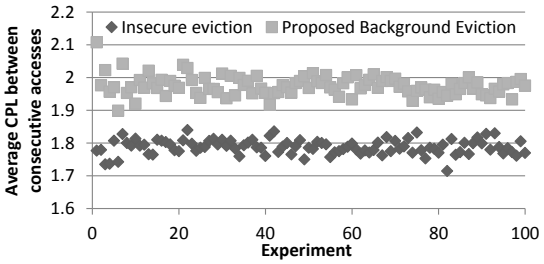

Figure 4: Average Common Path Length between consecutively-accessed paths with the insecure eviction scheme and the proposed background eviction scheme. This attack compromises the insecure eviction scheme.

There are other eviction schemes that are harder to break. For example, an eviction scheme can randomly remap one of the blocks in the stash and access a random path. This scheme creates no dependency between consecutive accesses and thus can defeat the above attack. However, it still tends to remap blocks from congested paths to less congested paths, and hence may be broken through more sophiscated attacks. We do not discuss these schemes and the corresponding attacks since they are tangential to the main topic of this paper.

3.1.4 Impact on performance

We modify the definition of access overhead defined in Section 2.4, taking into account dummy accesses introduced by background eviction,

$$Access\_Overhead = \frac{RA + DA}{RA} \frac{2(L + 1)M}{B} \quad (1)$$

where RA is the number of real accesses and DA is the number of dummy accesses. The access overhead of hierarchical Path ORAM is defined similarly as

$$Access\_Overhead = \frac{RA + DA}{RA} \frac{\sum_{i=1}^{H} 2(L_i + 1)M_i}{B_1}. \quad (2)$$

Though dummy accesses waste some cycles, background eviction now allows more efficient parameter settings that were previously prohibited due to high failure probability. We will show that background eviction improves the overall performance in Section 4.1.

3.2 Super Blocks

So far, we have tried to reduce the amount of data moved per ORAM access through background eviction and design space exploration. Another way to improve Path ORAM's efficiency is to increase the amount of useful data per ORAM access, by loading multiple useful blocks on an ORAM access. However, this is almost impossible in the original Path

ORAM, since blocks are randomly dispersed to all the leaves and are unlikely to reside on the same path.

To load a group of blocks on a single access, these blocks have to be intentionally mapped to the same leaf in the ORAM tree. We call such a group of blocks a *super block*. It is important to note that the blocks within a super block S do *not* have to reside in the same bucket. Rather, they only have to be along the same path so that an access to any of them can load all the blocks in S.

When a block $b \in S$ is evicted from on-chip cache, it is put back into the ORAM stash without waiting for other blocks in S. At this point it can find its way to the ORAM tree alone. When other blocks in S get evicted from on-chip cache at a later time, they will be assigned and evicted to the same path as b. We remark that this is the reason why super blocks are not equivalent to having larger blocks (cache lines): a cache line is either entirely in on-chip cache, or entirely in main memory.

3.2.1 Merging scheme

Super blocks create other design spaces for Path ORAM, such as super block size, which blocks to merge, etc. In this paper, we only merge adjacent blocks in the address space into super blocks. We believe this can exploit most of the spatial locality in an application, while keeping the implementation simple. We use the following simple scheme.

Static merging scheme: only merge adjacent blocks (cache lines) into super blocks of a fixed size. The super block size is determined and specified to the ORAM interface before the program starts. At initialization stage, data blocks are initially written to ORAM, and the ORAM interface simply assigns the same leaf label to the blocks from the same super block. The additional hardware required is small.

We believe dynamically splitting/merging super blocks based on temporal locality would yield better performance. We leave such schemes to future work.

3.2.2 Security of super blocks

For the same reasons as background eviction, an access to a super block must be indistinguishable from an access to a normal block for security reasons. In our scheme, a super block is always mapped to a random leaf in the ORAM tree in the same way as a normal block. If any block in the super block is accessed, all the blocks are moved from ORAM to on-chip cache and also remapped to a new random leaf. A super block access also reads and writes a path, which is randomly selected at the previous access to this super block. This is exactly the Path ORAM operation. Splitting/merging super blocks is performed on-chip and is not revealed to an observer.

3.2.3 Impact on performance

The super block scheme improves the access overhead by a factor of $|S|$ (the super block size in terms of blocks), if the $|S|$ blocks returned to the processor on a single access do have locality. In this case, performance is improved by reducing on-chip cache miss rate. The overhead of super blocks is more dummy accesses. For example, statically merging super blocks of size $|S|$ has similar effects as reducing the Z by a factor of $|S|$. We investigate the potential performance gain of super blocks in Section 4.3.

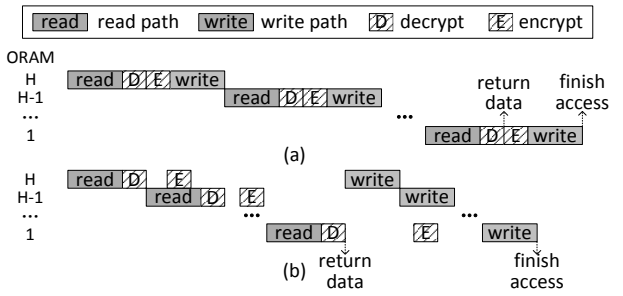

Figure 5: The access order in (b) hides encryption latency and can return data earlier than (a).

3.3 Other Optimizations

3.3.1 Exclusive ORAM

When connected to a processor, we design ORAM to be exclusive, i.e., any block in on-chip cache is not in ORAM. When a block (cache line) in ORAM is requested, the ORAM interface returns the block to on-chip cache and removes the block from ORAM. If the target block belongs to a super block, then the entire super block is removed from ORAM and put into on-chip cache. This guarantees that ORAM never has a stale copy of a block. So when a dirty cache line is evicted from on-chip cache, it can be directly put into the Path ORAM's stash without accessing any path. In contrast, if the ORAM is inclusive, it may contain a stale copy of the evicted block (if the block has been modified by the processor). Then the ORAM interface needs to make an access to update the stale copy in the ORAM.

Let us consider the scenario that most favors an inclusive ORAM: a program that scans memory and never modifies any block (all blocks are read-only). In this case, all the blocks evicted from the last-level cache are clean. In the inclusive ORAM, each last-level cache miss randomly remaps the requested block. In the exclusive ORAM, the requested block is removed after being read into the stash. Another block, which is mapped to a random leaf, is evicted from the last-level cache and put into the stash to make space for the requested block. So the inclusive ORAM and exclusive ORAM *still* add the same number of blocks to the stash, per access. Note that an exclusive ORAM should perform better when there are dirty cache lines evicted from on-chip cache.

It is also worth pointing out that in a conventional processor, a DRAM access is required if a dirty block needs to be written back to main memory. In that case, the access overhead of Path ORAM would be lower than what we defined in Equations 1 and 2.

3.3.2 Hierarchical ORAM access order

For a hierarchical ORAM $\{ORAM_1, ORAM_2, \ldots, ORAM_H\}$ ($ORAM_1$ is the data ORAM and $ORAM_H$ is the smallest position map ORAM), instead of performing accessHORAM() as described in Section 2.3 where each ORAM is read and written one by one, we propose the following more efficient access order, shown in Figure 5.

We first read a path from each ORAM, starting from $ORAM_H$ to $ORAM_1$. While reading $ORAM_h$, the ORAM interface re-encrypts blocks in $ORAM_{h+1}$'s stash and prepares those blocks to be written back. When $ORAM_1$ is read, the program data of interest is forwarded to the proces-

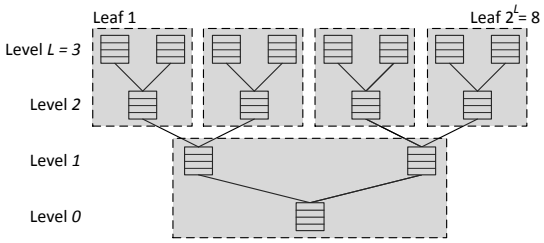

Figure 6: Illustration of subtree locality.

sor. Finally, the ORAM interface performs path writeback operations for each ORAM.

This strategy is better than a read/write each ORAM one by one strategy in the following two aspects. First, it hides the encryption latency by postponing the path writes, leaving the ORAM interface enough time to re-encrypt the blocks to evict. Second, the ORAM interface can return the requested data block after decrypting $ORAM_1$ so that the processor can start making forward progress earlier. Note that ORAM throughput does not change: the writeback operations must occur before the next set of reads.

3.3.3 Block size for position map ORAMs

Although we set the block size of data ORAM to be 128 bytes to utilize spatial locality in the programs, all the position map ORAMs should use a smaller block size, since all we need from a position map ORAM on an access is a leaf label, which is typically smaller than 4 bytes. The trade-off of a smaller block size is that as block size decreases, the number of blocks increases and hence the new position map grows, and thus more ORAMs may be needed to reduce the final position map to fit in on-chip storage.

The block size for position map ORAMs should not be too small because in that case: (1) other storage in a bucket (the addresses, the leaf labels and the 64-bit counter) dominate; and (2) adding an ORAM into the hierarchy introduces one decryption latency and one DRAM row buffer miss latency as discussed in Section 3.3.2.

3.3.4 Building Path ORAM on DRAM

Previous work on ORAM (and this paper so far) only focused on ORAM's theoretical overhead, the amount of extra data moved per access or similar metrics. ORAM has to be eventually implemented on commodity DRAM in order to be used in secure processors. However, if not properly done, Path ORAM can incur significantly larger overhead than the theoretical results when actually built on DRAM. For example, DRAM latency is much higher on a row buffer miss than on a row buffer hit. When naïvely storing the Path ORAM tree into an array, two consecutive buckets along the same path hardly have any locality, and it can be expected that row buffer hit rate would be low. We propose an efficient memory placement strategy to improve Path ORAM's performance on DRAM.

We pack each subtree with k levels together, and treat them as the nodes of a new tree, a 2^k-ary tree with $\lceil \frac{L+1}{k} \rceil$ levels. Figure 6 is an example with $k = 2$. We adopt the address mapping scheme in which adjacent addresses first differ in channels, then columns, then banks, and lastly rows. We set the node size of the new tree to be the row buffer size times the number of channels, which together with the original bucket size determines k.

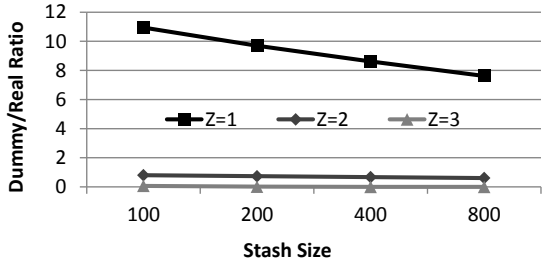

Figure 7: Dummy Accesses / Real Accesses vs. Stash Size (in blocks) in a 4 GB Path ORAM with 2 GB working set.

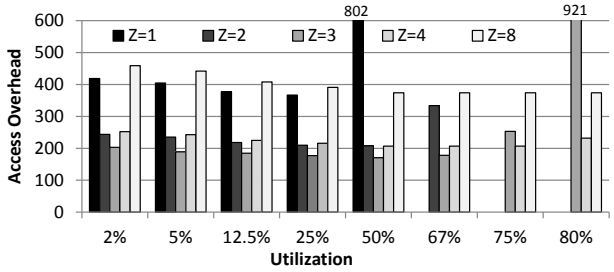

Figure 8: Access overhead of different ORAM sizes for 2 GB working set (e.g., 25% utilization corresponds to 8 GB ORAM).

4. EVALUATION

4.1 Path ORAM Design Space Exploration

We first explore the design space of Path ORAM to find configurations that minimize access overhead in Equations 1 and 2. Background eviction plays an important role: since it eliminates stash overflow, we can perform design space exploration with a single metric without worrying about Path ORAM failure probability.

4.1.1 Methodology

In all experiments, we use a 128-byte cache line size (= the block size for the data ORAM), the counter-based randomized encryption in Section 2.2.2, and assume that each bucket is padded to a multiple of 64 bytes. In each experiment, $10 \cdot N$ (N is the number of blocks in the ORAM) random accesses are simulated, excluding dummy accesses by background eviction.

4.1.2 Stash size

Figure 7 shows that for $Z \geq 2$, the percentage of dummy accesses is low to start, and only drops slightly as the stash size increases from 100 blocks to 800 blocks. The percentage of dummy accesses for $Z = 1$ is high—making $Z = 1$ a bad design point. We set stash size $C = 200$ for the rest of the evaluation.

4.1.3 Utilization

In Figure 8 we fix the working set (amount of valid data blocks) to be 2 GB and explore how other parameters impact access overhead. ORAM utilization is defined as the number of valid data blocks out of the N total blocks in the ORAM. This metric is directly related to DRAM utilization from Section 2.5.3.

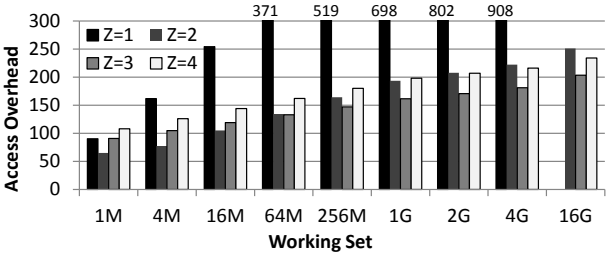

Figure 9: Access overhead of different ORAM sizes at 50% Utilization.

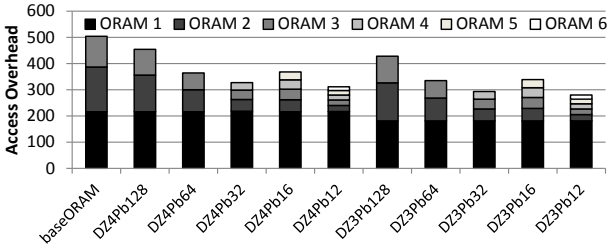

Figure 10: Overhead breakdown for 8 GB hierarchical ORAMs with 4 GB working set. DZ3Pb12 means data ORAM uses Z=3 and position map ORAMs have 12-byte block. The final position map is smaller than 200 KB.

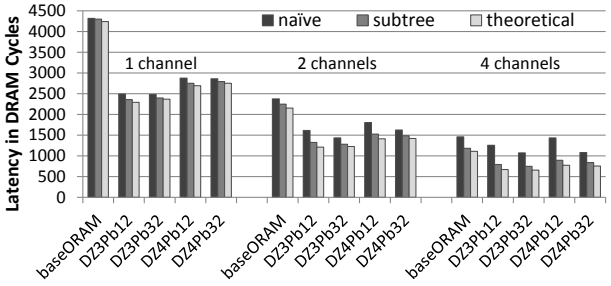

Figure 11: Hierarchical ORAM latency in DRAM cycles assuming 1/2/4 channel(s). Same notations as with Figure 10.

The best performance occurs at $Z = 3$ with 50% utilization. Access overhead increases slightly when utilization is too low because the path gets longer. But it is not very sensitive to low utilization since doubling the ORAM size only increases the path length by one. Access overhead also increases when utilization is too high because high utilization results in lots of dummy accesses. Smaller Z configurations (e.g., $Z = 1, 2$) are more sensitive to high utilization: their performance deteriorates more rapidly as utilization goes up. In fact, we do not have the results for $Z = 1$ at utilization $\geq 67\%$ or $Z = 2$ at utilization $\geq 75\%$ because these configurations are so inefficient that we cannot finish $10 \cdot N$ accesses for them.

$Z = 3$ at 67% utilization and $Z = 4$ at 75% utilization still have reasonable performance. This shows that the $1/Z$ utilization suggested in [21] was too pessimistic.

4.1.4 ORAM capacity

Figure 9 sweeps ORAM capacity with utilization fixed at 50%. For ORAMs larger than 256 MB, $Z = 3$ achieves the best performance. As the ORAM shrinks in size, the amount of dummy accesses decreases, and the benefit of smaller Z begins to show. For ORAMs between 1 MB to 64 MB, $Z = 2$ has the lowest overhead. This suggests that smaller Path ORAMs should use smaller Z. Figure 9 also shows that Path ORAM has good scalability; latency increases linearly as capacity increases exponentially.

4.1.5 Position map ORAM block size

Figure 10 shows the benefits of a small block size for position map ORAMs. We vary the block size of position map ORAMs and give the overhead breakdown. For each configuration, the number of ORAMs is chosen to get a final position map less than 200 KB. In the figure, DZ3Pb12 means the data ORAM uses $Z = 3$ and position map ORAMs have 12-byte blocks. We show results with both $Z = 3$ and 4 for the data ORAM because static super blocks may need $Z = 4$ to reduce dummy accesses. We fix block size to be 128 bytes for data ORAMs and $Z = 3$ for position map ORAMs (except baseORAM). baseORAM is the configuration used in [3]: a 3-level hierarchical Path ORAM where all the three ORAMs use 128 byte blocks, assume $Z = 4$, and use the strawman encryption scheme (Section 2.2.1).

Note that buckets are padded to a multiple of 64 bytes. This is why a 16-byte block size does not achieve good performance: both 16-byte and 32-byte block sizes result in a bucket size of 128 bytes. The optimal block size for position map ORAMs seems to be 12 bytes, followed by 32 bytes. However, Section 4.2 will show that the 12-byte design turns out to have larger overhead when actually implemented since it requires two more levels of ORAMs in the hierarchy than the 32-byte one. 32-byte position map ORAM block size with $Z = 3, 4$ reduces access overhead by 41.8% and 35.0% compared with the baseline.

4.2 Path ORAM on DRAM

We use DRAMSim2 [17] to simulate ORAM performance on commodity DRAM. We assume the data ORAM is 8 GB with 50% utilization (resulting in 4 GB working set); position map ORAMs combined are less than 1 GB. So we assume a 16 GB DRAM. We use DRAMSim2's default DDR3_micron configuration with 16-bit device width, 1024 columns per row in 8 banks, and 16384 rows per DRAM-chip. So the size of a node in our 2^k-ary tree (Section 3.3.4) is $ch \times 128 \times 64$ bytes, where ch is the number of independent channels. We evaluate the four best configurations in Figure 10, i.e., data ORAM $Z = 3$ and 4, and 12-byte/32-byte position map ORAM blocks.

Figure 11 shows the data latency (not counting decryption latency) of hierarchical Path ORAMs using the naïve memory placement and our subtree strategy, and compares these with the theoretical value, which assumes DRAM always works at its peak bandwidth. The figure shows that ORAM can benefit from multiple independent channels, because each ORAM access is turned into hundreds of DRAM accesses. But this also brings the challenge of how to keep all the independent channels busy. On average, the naïve scheme's performance becomes 20% worse than the theoretical result when there are two independent channels and 60% worse when we have four. Our subtree memory placement strategy is only 6% worse than the theoretical value with two channels and 13% worse with four. The remaining overhead comes from the few row buffer misses and DRAM refresh.

Table 1: System configuration for the baseline and secure processor. On a cache miss, the processor incurs the cache hit plus miss latency.

Core model: in order, single issue	
Cycle latency per Arith/Mult/Div instr	1/4/12
Cycle latency per FP Arith/Mult/Div instr	2/4/10
Cache	
L1 exclusive I/D Cache	32 KB, 4-way
L1 I/D Cache hit+miss latencies	1+0/2+1
L2 exclusive Cache	1 MB, 16-way
L2 hit+miss latencies	10+4
Cache block size	128 bytes

Table 2: Path ORAM latency and on-chip storage of the configurations evaluated in Section 4.3. All cycle counts refer to CPU cycles.

ORAM config.	baseORAM	DZ3Pb32	DZ4Pb32
return data (cycles)	4868	1892	2084
finish access (cycles)	6280	3132	3512
stash size (KB)	77	47	47
position map size (KB)	25	37	37

Even though a 12-byte position map ORAM block size has lower theoretical overheads, it is worse than the 32-byte design.

We remark that it is hard to define Path ORAM's slowdown over DRAM. On one hand, DDR3 imposes a minimum ~ 26 (DRAM) cycles per access, making Path ORAM's latency $\sim 30\times$ over DRAM assuming 4 channels. On the other hand, our Path ORAM consumes almost the entire bandwidth of all channels. Its effective throughput is hundreds of times lower than DRAM's peak bandwidth ($\approx$ access overhead). But the actual bandwidth of DRAM in real systems varies greatly and depends heavily on the applications, making the comparison harder.

4.3 Path ORAM in Secure Processors

We connect Path ORAM to a processor and evaluate our optimizations over a subset of the SPEC06-int benchmarks. The processors are modeled with a cycle-level simulator based on the public domain SESC [16] simulator that uses the MIPS ISA. Instruction/memory address traces are first generated through SESC's rabbit (fast forward) mode and then fed into a timing model that represents a processor chip with the parameters given in Table 1. Each experiment uses SPEC reference inputs, fast-forwards 1 billion instructions to get out of initialization code and then monitors performance for 3 billion instructions.

Table 2 lists the parameters for configurations DZ3Pb32, DZ4Pb32 and baseORAM, including the latency (in CPU cycles) to return data and finish access, as well as the on-chip storage requirement for the stash and position map. Assuming CPU frequency is 4× of DDR3 frequency,

$$latency_{CPU} = 4 \times latency_{DRAM} + H \times latency_{decryption}.$$

We also compare against a conventional processor that uses DRAM. Path ORAMs and DRAMs are both simulated using DRAMSim2.

Figure 12 shows the SPEC benchmark running time using different Path ORAM configurations and super blocks, normalized to the insecure processor with DRAM. DZ3Pb32

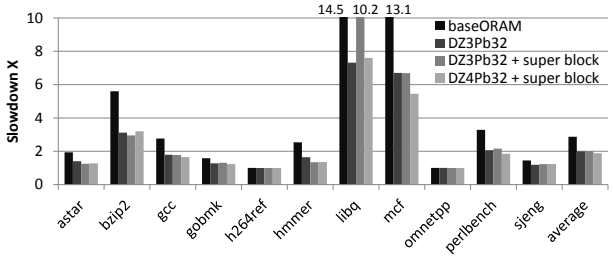

Figure 12: SPEC benchmark performance with optimized Path ORAM.

reduces the average execution time by 43.9% compared with the baseline ORAM. As expected, the performance improvement is most significant on memory bound benchmarks (mcf, bzip2 and libquantum).

In this experiment, we only statically form super blocks of size two (consisting of two blocks). On average, DZ4Pb32 with super blocks outperforms DZ3Pb32 without super blocks (the best configuration without super blocks) by 5.9%, and is 52.4% better than the baseline ORAM. There is a substantial performance gain on applications with good spatial locality (e.g., mcf) where the prefetched block is likely to be accessed subsequently. Using static super blocks with DZ3Pb32 slightly improves the performance on most benchmarks, but has worse performance on certain benchmarks because it requires too many dummy accesses, canceling the performance gain on average.

5. INTEGRITY VERIFICATION

Orthogonal to performance optimizations, we can build an integrity verification layer on top of Path ORAM so that a secure processor can verify that the retrieved blocks from Path ORAM are *authentic*, i.e., they were produced by the secure processor, and *fresh*, i.e., a block in the ORAM corresponds to the latest version that the processor wrote.

A strawman approach to implementing the integrity layer is to store a Merkle tree in external memory. Each leaf of the Merkle tree stores a 128-bit hash of a data block in the ORAM. We note that this scheme would work with any kind of ORAM, and similar ideas are used in [14]. To verify a block, a processor needs to load its corresponding path and siblings in the Merkle tree and check the consistency of all the hash equations. This scheme has large overheads for Path ORAM, because all the $Z(L+1)$ data blocks on a path have to be verified on each ORAM access. So $Z(L+1)$ paths through the Merkle tree must be checked per ORAM access, which contain $Z(L+1)^2$ hashes in total. (Z and L are given in Section 2.1.)

To implement integrity verification with *negligible* overhead, we exploit the fact that *the basic operation of both the Merkle tree and Path ORAM is reading paths through their tree structures.* In particular, we create an authentication tree that has exactly the same structure with Path ORAM (shown mirrored in Figure 13). (We describe ideas and give an example here; additional details can be found in [4].)

To avoid having to initialize the authentication tree at program start time,[4] we add two bits to each bucket—labeled

[4]We assume that at start-up time, the authentication and ORAM trees consist of random bits corresponding to the uninitialized DRAM state.

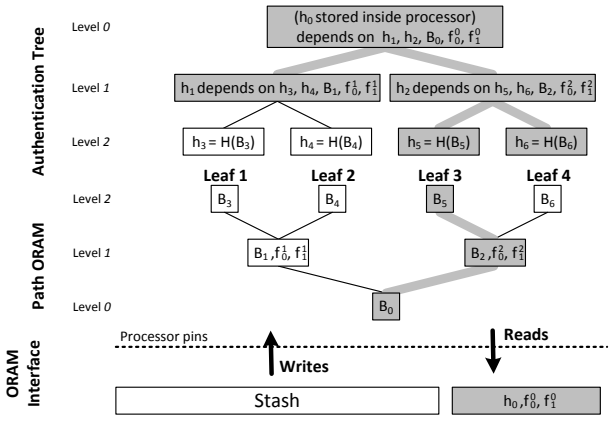

Figure 13: Integrity verification of Path ORAM.

f_0^i and f_1^i for bucket i and stored in external memory along with bucket i—that are conceptually valid bits for bucket i's children. We say bucket i is *reachable* from the root bucket if all valid bits on the path, from the root bucket to bucket i, equal 1. We define $\mathsf{reachable}(B_i) = 1$ if B_i was reachable at the start of a particular ORAM access and $= 0$ otherwise. We maintain the invariant that all reachable buckets from the root bucket have been written to through ORAM operations at some point in the past.

Each intermediate node in the authentication tree now stores the hash of the concatenation of (a) child bucket valid flags, (b) the *corresponding bucket* in the Path ORAM tree, and (c) the sibling hashes for that intermediate node. Authentication works as follows: Suppose the root bucket is labeled B_0 and the root hash/child valid flags (stored inside the ORAM interface) are $h_0/f_0^0/f_0^1$ respectively. We initialize $h_0 = H(0)$ and $f_0^0 = f_1^0 = 0$ at program start time. Following the figure: to perform an ORAM access to block B_5 mapped to leaf $l = 3$, the ORAM interface performs the following operations:

1. ORAM path read: read B_0, B_2 and B_5 and child valid flags f_0^2, f_1^2.
2. Read sibling hashes for the path (h_1 and h_6).
3. Compute $h'_5 = H(B_5)$, $h'_2 = H(f_0^2||f_1^2||(f_0^2 \vee f_1^2) \wedge B_2||f_0^2 \wedge h'_5||f_1^2 \wedge h_6)$, $h'_0 = H(f_0^0||f_1^0||(f_0^0 \vee f_1^0) \wedge B_0||f_0^0 \wedge h_1||f_1^0 \wedge h'_2)$, where '$\vee$' and '$\wedge$' are logical OR/AND operators.[5]
4. If $h_0 = h'_0$, the path is authentic and fresh!
5. Update child valid flags: $f_0^{0\prime} = f_0^0$, $f_1^{0\prime} = f_0^{2\prime} = 1$ and $f_1^{2\prime} = f_1^2 \wedge \mathsf{reachable}(B_2)$. Update the root bucket child valid flags (inside the ORAM interface) to $f_0^{0\prime}$,$f_1^{0\prime}$.
6. ORAM path writeback: evict as many blocks as possible from the stash to the path 3 (forming B'_0, B'_2 and B'_5). Write $f_0^{2\prime}$,$f_1^{2\prime}$ as the new child valid flags for B'_2.
7. Re-compute h_5, h_2 and h_0; write back h_5 and h_2.

All data touched in external memory is shaded in Figure 13.

Note that only the sibling hashes need to be read in from the authentication tree. The hashes on the path of interest are computed by the processor, by hashing the buckets read via the Path ORAM operation concatenated to the sibling hashes. We point out that since hashes are computed from the leaves to the root, only the reachable portion of the path in the authentication tree needs to be read per access. That is, if the path to B_5 is being accessed (see above) and $f_0^0 = f_1^0 = 0$ at the time of the access, $h'_0 = H(0||0||0||0||0) = H(0)$, which is independent of any values in the authentication tree. Conceptually, the child valid flags indicate a frontier in the ORAM/authentication trees that has been touched at an earlier time.

In summary, on each ORAM access at most $L \ll (L+1)^2 Z$ (sibling) hashes need to be read into the processor and L hashes (along the path) need to be written back to the external authentication tree. This operation causes low performance overhead beyond accessing ORAM.

[5]Note that $(f_0^i \vee f_1^i) \wedge B_i = B_i$ if $\mathsf{reachable}(B_i) = 1$ and is only needed to get the correct value for h'_0 before the first access is made. This OR-AND operation is applied to other non-leaf buckets for the sake of consistency, but is not required.

6. RELATED WORK

6.1 Secure Hardware

The TPM [25, 1, 18] is a small chip soldered onto a motherboard capable of performing a limited set of secure operations; the TPM assumes trust in the OS, RAM, Disk, connecting bus and the user application. The TPM, and user systems such as Intel's TPM+TXT [9], do not consider address bus leakage in their threat model and therefore ORAM can be used in conjunction with them to achieve higher security. Aegis [23, 24] is a single-chip processor and the first to provide memory integrity verification and encryption, which allows memory to not be trusted, but assumes trust in the OS kernel and user application. eXecute Only Memory (XOM) [11, 12, 13] trusts only the user application and the processor chip but needs to be protected against replay attacks. Aegis and XOM need additional functionality to be protected against attacks based on memory access patterns of badly-written programs or programs with bugs, and our work in this paper can be used to guarantee security in these scenarios.

The Trusted Execution Module TEM [2] is a secure coprocessor capable of securely executing partially-encrypted procedures/closures expressing arbitrary computations which fit inside the TEM. ORAM would enable TEM to use external memory without sacrificing security.

Ascend [3] (followed up in the more comprehensive [4]) uses Path ORAM to perform encrypted computation assuming untrusted programs. We have compared our ORAM configurations and associated architectural optimizations to those used in a preliminary publication on Ascend [3] and shown significant improvements. We note again that Ascend performs ORAM accesses strictly periodically which increases overhead slightly; our focus here is on the efficiency of the ORAM primitive as opposed to a specific usage scenario, and therefore we have not assumed periodic accesses for the Ascend baseline or for our ORAMs.

6.2 Memory Access Pattern Leakage

HIDE [30] (and follow-on work [5], [27]) has architectural support to obfuscate memory access patterns through the idea of randomly shuffling memory locations between consecutive accesses (similar to ORAM). However, to have small performance overheads, HIDE only applies this technique within small chunks of memory (usually 8 KB to 64 KB). In our threat model, obfuscation over small chunks breaks security because the server can engineer a data placement for

a client program, or engineer a curious program to perform inter-chunk accesses based on private data, and decipher all the encrypted data. If HIDE were to apply shuffling over 4 GB memory, the on-chip storage requirements would correspond to an untenable amount of cache memory on-chip. Therefore, to achieve cryptographic-grade security, it is much better to use our optimized Path ORAM.

7. CONCLUSIONS

Traffic from processor chip pins to DRAM memory is a side channel that can be easily monitored by software, and is hard to enclose in a tamper-resistant package. Therefore, it is important to thwart attackers who try to exploit this side channel to discover private data. Oblivious RAM can guarantee security by blocking this side channel; this requires that an ORAM interface be built into the chip, which increases the amount of off-chip traffic logarithmically in the worst case.

"Default" configurations of Path ORAM can result in over 10× performance degradation for benchmarks such as SPEC if all data is considered private. Through novel architectural mechanisms such as background eviction and super blocks, as well as comprehensive design space exploration, we have shown that this overhead can be significantly reduced. Ongoing improvements include using different kinds of ORAMs for streaming data [29] and optimizing benchmarks with high locality using dynamic super block schemes.

8. REFERENCES

[1] W. Arbaugh, D. Farber, and J. Smith, "A Secure and Reliable Bootstrap Architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997, pp. 65–71. [Online]. Available: citeseer.nj.nec.com/arbaugh97secure.html

[2] V. Costan, L. F. G. Sarmenta, M. van Dijk, and S. Devadas, "The trusted execution module: Commodity general-purpose trusted computing," in *CARDIS*, 2008.

[3] C. Fletcher, M. van Dijk, and S. Devadas, "Secure Processor Architecture for Encrypted Computation on Untrusted Programs," in *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing*, Oct. 2012, pp. 3–8.

[4] C. W. Fletcher, "Ascend: An architecture for performing secure computation on encrypted data," in *MIT CSAIL CSG Technical Memo 508*, April 2013. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-508/Memo-508.pdf

[5] L. Gao, J. Yang, M. Chrobak, Y. Zhang, S. Nguyen, and H.-H. S. Lee, "A low-cost memory remapping scheme for address bus protection," in *Proceedings of the 15th PACT*, ser. PACT '06. ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1152154.1152169

[6] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and Merkle Trees for Efficient Memory Integrity Verification," in *Proceedings of Ninth International Symposium on High Performance Computer Architecture*. New-York: IEEE, February 2003.

[7] O. Goldreich, "Towards a theory of software protection and simulation on oblivious rams," in *STOC*, 1987.

[8] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *J. ACM*, 1996.

[9] D. Grawrock, *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[10] R. Huang and G. E. Suh, "Ivec: off-chip memory integrity protection for both security and reliability," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10, 2010, pp. 395–406.

[11] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz, "Specifying and verifying hardware for tamper-resistant software," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[12] D. Lie, C. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.

[13] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, November 2000, pp. 168–177.

[14] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman, "Toward practical private access to data centers via parallel oram." *IACR Cryptology ePrint Archive*, vol. 2012, p. 133, 2012, informal publication. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2012.html#LorchMPRS12

[15] R. Ostrovsky, "Efficient computation on oblivious rams," in *STOC*, 1990.

[16] J. Renau, "Sesc: Superescalar simulator," university of illinois urbana-champaign ECE department, Tech. Rep., 2002. [Online]. Available: http://sesc.sourceforge.net/index.html

[17] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[18] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS," in *Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC'06)*, Nov. 2006.

[19] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with $o((\log n)^3)$ worst-case cost," in *Asiacrypt*, 2011, pp. 197–214.

[20] E. Stefanov and E. Shi, "Path O-RAM: An Extremely Simple Oblivious RAM Protocol," Cornell University Library, arXiv:1202.5150v1, 2012, arxiv.org/abs/1202.5150.

[21] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," in *NDSS*, 2012.

[22] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors," in *Proceedings of the* 36^{th} *Int'l Symposium on Microarchitecture*, Dec 2003, pp. 339–350.

[23] ——, " AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*. New-York: ACM, June 2003. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-474/Memo-474.pdf(revisedone)

[24] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of the* 32^{nd} *Annual International Symposium on Computer Architecture*. New-York: ACM, June 2005. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf

[25] Trusted Computing Group, "TCG Specification Architecture Overview Revision 1.2," http://www.trustedcomputinggroup.com/home, 2004.

[26] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, ser. ISCA '06, 2006, pp. 179–190.

[27] J. Yang, L. Gao, Y. Zhang, M. Chrobak, and H. Lee, "A low-cost memory remapping scheme for address bus protection," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 443–457, 2010.

[28] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, dec. 2003, pp. 351 – 360.

[29] X. Yu, C. Fletcher, L. Ren, M. van Dijk, and S. Devadas, "Efficient private information retrieval using secure hardware," in *MIT CSAIL CSG Technical Memo 509*, April 2013. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-509/Memo-509.pdf

[30] X. Zhuang, T. Zhang, and S. Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 2004, pp. 72–84.